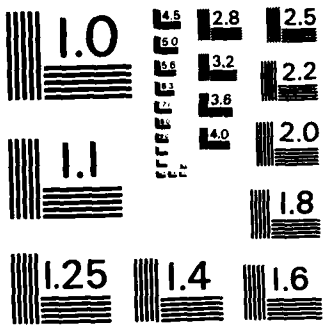
AD-A160 451 GUIDELINES FOR A MINIMAL ADA RUNTIME ENVIRONMENT(U) 1/1
SOFTECH INC WALTHAM MA V GROVER JAN 85 3285-2-208/5
ESD-TR-85-139 F33600-84-D-0280

UNCLASSIFIED F/G 9/2 NL

END
FILMED
DTIC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

(12)

ESD-TR-85-139 3285-2-208/5

AD-A160 451

GUIDELINES FOR A MINIMAL ADA RUNTIME ENVIRONMENT

VINOD GROVER

SOFTECH, INC
460 TOTTEN POND ROAD
WALTHAM, MA 02254

JANUARY 1985

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

PREPARED FOR

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR ACQUISITION LOGISTICS
AND TECHNICAL OPERATIONS
HANSCOM AIR FORCE BASE, MASSACHUSETTS 01731

85 10 22 020

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

[illegible]
ANTHONY L. STEADMAN
Project Officer, Project 2526
Software Engineering Tools and Methods

[illegible]
WILLIAM J. LETENDRE
Program Manager,
Computer Resource
Management Technology

FOR THE COMMANDER

[illegible]
ROBERT J. KENT
Director, Computer Systems Engineering
Deputy for Acquisition Logistics
and Technical Operations

[illegible]
ROBERT G. HOWE (MITRE)
Group Leader

nclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS AD-A160 451
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 3285-2-208/5	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-85-139
6a. NAME OF PERFORMING ORGANIZATION SofTech, Inc / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Hq Electronic Systems Division (AL)
6c. ADDRESS (City, State and ZIP Code) 460 Totten Pond Road Waltham, MA 02254	7b. ADDRESS (City, State and ZIP Code) Hanscom Air Force Base, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Deputy for Acquisition / 8b. OFFICE SYMBOL (If applicable) ESD/AL	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33600-84-D-0280
8c. ADDRESS (City, State and ZIP Code) Electronic Systems Division Hanscom Air Force Base, MA 01731	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. 5720 / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) Guidelines for a Minimal Ada Runtime Environment	
12. PERSONAL AUTHOR(S) Vinod Grover	
13a. TYPE OF REPORT Final Report / 13b. TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Yr., Mo., Day) 1985 January / 15. PAGE COUNT 58
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Ada Runtimes Environment

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A major goal of the JAMPS Ada software acquisition is to develop highly portable and reusable modules for JINTACCS message preparation/handling that can easily be used in many different types of military systems. This goal impacts the selection and use of the Ada runtime environment, as dependence on features of the runtime environment (either for functionality or performance) limits portability to only those systems providing the same features. This results in the concept of a minimal set of runtime features necessary to support JAMPS; this minimal set, as identified herein, is potentially applicable to other real time systems.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. [✓ X] DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

EXECUTIVE SUMMARY

The study is intended to define the characteristics of an Ada Runtime Environment that effectively supports the development of real-time applications. The study was undertaken with the specific objective of supporting the selection of a runtime environment for the JINTACCS Automated Message Preparation System (JAMPS) program, but provides results useful in any real-time system acquisition program.

Ada [illegible]

A major goal of the JAMPS acquisition is to develop highly portable and reusable implementation that can be used in many different military systems. This goal impacts the selection and use of the runtime environment, as dependence on particular features of the runtime environment (either for functionality or performance) limits portability to only those systems providing the same features. This results in the concept of a minimal set of runtime environment features necessary to support JAMPS. Implementing JAMPS with dependence only on these features will allow portability within the class of systems providing those features. This report identifies such a minimal set of environment features. Keywords: Fault tolerant computing, [illegible] output processing, data [illegible] systems.

This report is intended for use by two main audiences:

a. Procuring agencies or contractors wishing to select an Ada implementation that meets the needs of a particular real-time program.

b. Ada compiler implementors wishing to develop an Ada implementation that is responsive to the needs of real-time programs.

ACKNOWLEDGEMENT

This document was prepared by SofTech, Inc., under Air Force contract F33600-84-D-0280 (administered by Ms. Pat Knoop, Language Control Facility, Wright-Patterson Air Force Base). Financial support was received from Program Element 64760F, Embedded Computer Systems Engineering and Applications, Electronic Systems Division, United States Air Force Systems Command, Hanscom Air Force Base (ESD/ALSE). The MITRE Corporation provided technical guidance.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

SECTION 1

INTRODUCTION

A runtime support environment (RSE) for a programming language usually provides support for the generated code in the target environment. The target environment may be a bare machine or another virtual machine (e.g., an operating system). Usually there are certain high-level language features that cannot easily be mapped onto the target environment support (i.e., instruction set, system calls etc.). The runtime system comprises those procedures which implement such language features.

1.1 THE ROLE OF RUNTIME SYSTEMS IN APPLICATIONS

From the user's point of view, a runtime system has a different meaning. For embedded systems applications, a runtime system is viewed as a substitute for an operating system. This implies that not only should it provide the appropriate functionality, but must provide it efficiently. Furthermore, in the light of portability, such functionality and efficiency must be easily provided for in many different candidate target systems.

Currently, within the user community there are several distinct concerns about the role of runtime systems in applications [6]. In the following sections we discuss some of these concerns.

1. There are many features of the Ada language that are not sufficiently well defined or are implementation defined. This gives the implementor of the language a wide variety of options. The choice of one or a particular set of options is dependent on the implementor's convenience. However, the choice of an option is certainly bound to affect an application. To give an example, consider an implementation of the task scheduling strategy that switches tasks around when they run to completion or voluntarily wait on some event. (Under this implementation, if there are certain tasks that execute a perpetual loop without reliance on external events, then once one such task is executing others will be permanently locked out.) Under a time-sliced implementation, the effect of the same program would be different.

2. There are certain well defined areas of the language that have a wide variety of different implementations. The only

foreseeable effect of the choice of an implementation technique may have to do with the efficiency of an application program.

3. There may be well defined capacity limitations imposed by the implementations which restrict the static or the runtime behavior of the application program. This category includes items such as the number of lines in a package and the maximum amount of dynamic storage available to a program or the subprogram calling depth. Such concerns will at best restrict the portability of the application program and at worst expose surprises at runtime. For example, a particularly restrictive implementation may raise a STORAGE ERROR after the call depth becomes greater than 20. This behavior may have unexpected consequences.

4. Certain applications cannot best be implemented in Ada, with all of its underlying runtime and compile time machinery. They require some additional support, usually outside Ada, from the implementation. To give a concrete example, consider an application which is required to perform, among other things, built in read/write memory tests. Even though these tests can be performed using Ada, by resorting to low level facilities like representation clauses, it is not considered safe to do so, since such destructive tests could interfere with the runtime data and code areas. Such tests can best be performed by some cooperation or support from the runtime system.

5. Many of the concerns are largely based on the fact that the application programmers are reluctant to use Ada and do not understand its use. Many of the Ada language features, are sufficiently high level to make these programmers suspicious, and they ask, for example, how many programs can simultaneously be executed by a particular Ada implementation. These concerns can mostly be dismissed by proper education and a demonstration of the adequacy of Ada features.

1.2 SCOPE OF THIS WORK

With all these concerns and uncertainties about the language and its runtime environment, the runtime system is viewed, from the application designer's perspective, as possessing certain features. The presence or absence of these features, other than those required by the language, may yield widely varying application designs. Ultimately, this may mean that one particular design may or may not run as intended on a given implementation of the language. The purpose of this report is to describe a set of runtime environment features which are highly suitable for the design and implementation of the JINTACCS* Automatic Message Preparation System (JAMPS). Ideally, one can safely define such a set of features if the final design alternatives are known. However, in this case we are only partly familiar with the current design of JAMPS, which has been implemented in "C" language.

The starting point of this report is the list of concerns raised by a straightforward reimplementation of the current JAMPS design, in Ada, on a target environment devoid of the operating system. This list of concerns has been raised by R. G. Howe of the MITRE Corporation as a set of questions regarding the capabilities of an Ada runtime environment. In this report we analyze some of the concerns and try to see how they might arise in the design of JAMPS. Having established a need for some of the concerns, we try to determine if such concerns should be addressed by an Ada runtime environment; the chief criteria being portability, efficiency, and ease of implementation of the proposed features. We have found that most of the concerns which do not involve the capacity of the runtime system can be addressed almost completely in Ada, with minimal additional support from the runtime system.

From this analysis, a set of tentative guidelines is derived for features which should be an essential part of an Ada implementation. For the sake of completeness, we also included a list of similar concerns raised during the design of certain other applications, in the expectation that such concerns may arise during the reimplementation of JAMPS in Ada if a redesign is necessary.

1.3 ORGANIZATION OF THE REPORT

This report is organized as follows: In Section 2, we give a taxonomy of the various issues which concern the application programmer in general. We try to give a rationale and the context in which such issues arise. Many examples are taken specifically from JAMPS, and many others from other application systems we have encountered. These issues form the basis for suggesting the plausible features of an Ada runtime organization.

In Section 3, we present a preliminary analysis of these issues from the perspective of efficiency, portability, and implementation ease (both from the view of the user and the language implementor). This gives us a method for deciding the features of runtime environments which can and should be implemented for the JAMPS application. For the features which we feel should not be part of the runtime system, we describe, with examples, how they can be implemented in

* Joint Interoperability for Tactical Command & Control Systems (JINTACCS)

application code. In Section 4, we enumerate all the features of a minimum runtime environment relative to JAMPS.

Finally, in Section 5, we outline our conclusions and suggest some areas for further study.

SECTION 2

RUNTIME ENVIRONMENT ISSUES

Following is a list of typical user concerns, drawn mostly from the JAMPS application design.

2.1 MEMORY MANAGEMENT

2.1.1 Selective Linking

> If no access type is declared, then will the implementation still require some overhead? Or, in general, if a language feature X is not used, then will the implementation still require some overhead in terms of space or time?

This problem appears in all those cases, where general purpose (i.e., reusable) software (e.g., the runtime system) is used for a specific less general application. This occurs when only part of the general purpose package is actually required by the application. This means that all or most of the general purpose software is going to be loaded into the memory. This is clearly wasteful in embedded systems. For example, consider a general purpose stack package which provides operations POP and PUSH. If a certain application never uses POP but uses PUSH, then the code for POP should never be loaded. The same concern applies to the runtime system and the reusable software written by the user. This falls broadly under the category of selective linking.

2.1.2 Free Storage Management

> What storage management (allocation and deallocation) facilities are available in the underlying system? What is the allocation strategy? First-fit? Best-fit? Other? What deallocation strategy is used? Automatic or on demand? What are the limits of allocation for procedures, tasks, and packages?

The allocation and deallocation mechanisms are vital to many applications. Ada provides the use of the allocator mechanism new for allocating space. It is important for most embedded applications, including JAMPS, that the algorithms used for dynamic allocation be quite efficient. It is not very important to know the exact allocation strategy though. Even more important is the deallocation strategy used by the system. Automatic garbage collection may be desirable from the point of view of the programmer, since it allows him to be free of this concern. However, in most embedded systems it may mean that the time used by a certain section of the code becomes unpredictable. For certain time critical procedures it may be necessary to have the garbage collection turned off.

2.1.3 Free Storage Monitoring

Are there any methods for detecting the state of working storage? How can I find out if X percent of working storage is exhausted?

This problem is typical of embedded systems. Such systems which are often required to take certain actions when a certain fraction of free storage is used up. Typically, these systems will explicitly reclaim storage at this point or decide to allocate storage on disk.

Ada does not address this problem explicitly, except in the case when <u>pragma</u> STORAGE_SIZE is used for specifying the size of the heap associated with a collection. In the case of all storage being used up an exception STORAGE_ERROR is raised. JAMPS requires that when 80% of the system space is used up certain action be taken.

2.1.4 Free Storage Structuring

Is there one heap, or is a separate memory region reserved for each collection?

The ability to declare separate heaps is important to many applications. With this capability it may be easy to prove that a certain allocator in a time-critical region cannot raise the STORAGE ERROR exception. This leads to considerable simplifications including providing no handler and eliminating the check. Also, it may be desirable to put an upper bound on the storage available to a low priority function, thus ensuring that a higher priority function will not be deprived. Furthermore, the presence of separate heaps may also prevent contention among several tasks competing for storage. It would be pleasi..g to have the implementation associate a separate heap for a collection for which pragma STORAGE SIZE has been used. However, the only disadvantage is that excessive use of this capability may lead to excessive fragmentation and cause inefficiencies in the storage management functions.

2.1.5 Target Memory Control

Will the user be allowed to statically assign an appropriate amount of working storage at time of system generation (via LINKAGE EDITOR)? Does the implementation support the use of the pragma MEMORY_SIZE?

It is quite common, during the life cycle, to change the memory configuration available to an embedded system. The reason for this might be to improve the runtime performance. This could be used to increase the size of the runtime heap available to the application code. In the light of these requirements, it should be possible to redefine the memory regions available to the generated code and the runtime system. The Ada mechanism for doing this is the pragma MEMORY SIZE. This defines the memory size, in STORAGE_UNITS, of the target system, and is used by the Ada implementations to assign code and runtime areas.

2.1.6 Addressing Limitations

What are the limitations on memory space as occupied by individual packages, tasks, and subprograms? What are the absolute addressing limits of the code?

This is a problem that comes up regarding the usability of most language implementations, and affects portability in many ways. Depending on the choices taken by the compiler writers and the idiosyncrasies of the target machine several restrictions are placed on the programmer. We describe some such restrictions here, and the reader should also consult the separate report on Ada Portability Guidelines for further discussion on these issues.

Packages: Depending on the target machine, the size of the packages may be restricted to some specified maximum. For example, if library packages are represented as segments on an Intel 8086, then the code size of the package may be restricted to 64K. Further limits may be imposed on the size of the data that can be declared inside such packages, depending on the implementation strategy chosen; for instance if the package code, package constants, and package variables are represented in separate segments then the source size of packages can be quite large.

Subprograms The strategy chosen to represent subprograms at runtime will have an impact on the textual size, the call depth (or nesting), and the number and type of parameters that can be passed.

For Intel 8086 if a stack cannot grow beyond a single segment, or the size of a stack frame is limited to a segment, this would have several implications for the programmer.

Tasks: Similarly a task's runtime representation (i.e., its stack and the frame) would dictate the data areas and the depth of calls originating from that task.

The restrictions imposed by problems of this nature can in many cases (though not all) be circumvented. For instance, if a compiler does not accept a package with 20 procedures or with too many source lines, then it is a mechanical (but tedious) task to split up the package to satisfy the compiler. But if the implementation blows up because the stack overflows or there are too many task in the system, then it is a non-trivial problem to re-design the system to overcome this problem.

The only course left for the user is to design his system with these restrictions in mind, and if possible, know them in advance before any implementation is undertaken.

2.2 PROCESSOR MANAGEMENT

2.2.1 Size Of Task Population

What restrictions are placed on the maximum and minimum number of active tasks in a given application?

This is a very important design concern for many applications. Up to a point, the throughput of an application can be increased by increasing the level of multiprogramming. Systems with high throughput requirements will, in general, tend to have a larger number of active tasks than the ones with low throughput requirements. This may be so, even if the number of different task types is relatively small. To give an example, consider a message handling system (e.g. JAMPS). It may be desirable to assign a task for each message in transit; with all such tasks having the same type. Therefore, it is important for performance that the implementation allow a large number of tasks.

2.2.2 Task Priorities

Does the implementation support the use of pragma PRIORITY? If so what is the declaration of subtype priority range? Can any one task be invoked at more than one priority level? What algorithm is applied for tasks of undefined priority?

In many real-time embedded systems, the use of priorities is important in order to specify the relative urgency of various tasks. Based on experience and literature searches, the maximum number of priority levels in most systems is not likely to be greater than ten. Nevertheless, it would be pleasing if it was possible to change the range of priorities, and use system level priorities (if available). Ada does not allow one to invoke a task or task type at more than one priority level. The only means of 'changing' the priority level of a task is to engage it in a rendezvous with a task having a higher priority. The higher priority of the rendezvous only lasts for the duration of the rendezvous.

The tasks with undefined priority are immune to the priority rules as defined in the language. The intent behind this was to be able to define certain "server" tasks which do not have a priority of their own, e.g., bounded buffers. For such tasks the language implementors can sometimes take shortcuts for implementing rendezvous and activation (see [3]). Therefore, in keeping with the spirit of Ada, the users should not depend on the priority rules for unprioritized tasks.

2.2.3 Task Dispatching

Does the implementation support the use of expedited dispatching? What is it?

The tasking model of Ada is very expressive, and subsumes the traditional notions of interrupt handlers and bounded buffers used extensively in real-time systems. In fact, most of the I/O in a large number of systems is done via bounded buffers. A typical example of such a bounded buffer in Ada is described below, and an equivalent description using the concept of an interface module of MODULA [7] is given. A bounded buffer could be viewed as a data structure with two operations GET and PUT defined on it. The effect of a GET is to suspend the caller when the bounded buffer is EMPTY, and return some data when it is not. The effect of a PUT is to suspend the caller if the buffer is FULL. An Ada implementation of such a bounded buffer is to encapsulate the buffer within a task and the operations are implemented as entries.

As is evidenced from the above two examples, the Ada code is more elegant, but the equivalent MODULA code imposes less overhead. A naive Ada implementation may actually create a special task for such purposes when a simple semaphore or signal scheme can be used.

```
task BOUNDED BUFFER is
    entry GET(x : out DATA);
    entry PUT(x : in  DATA);
end BOUNDED BUFFER;

task body BOUNDED BUFFER is
    BUFFER : ...;
begin
    loop
        select
            when not EMPTY(BUFFER) =>
                accept GET(x : out DATA) do
                    DATA := ...;
                end GET;
        or
            when not FULL(BUFFER) =>
                accept PUT(x : in DATA) do
                -- store DATA in BUFFER
                end PUT;
        end select
    end loop;
 end BOUNDED BUFFER;
```

Figure 2-1. An Example of a Bounded Buffer in Ada

```
module  BOUNDED BUFFER;
define GET; PUT;
var buffer : ...;
    notempty, notfull : signal;

    function GET : DATA;
    begin
        await(NOTEMPTY);
        get := /* next available item from the buffer */
        send(NOTFULL);
    end GET;

    function PUT(x : DATA);
    begin
        await(NOTFULL);
        /*
         * store x at the next
         * available position in the BUFFER
         */
         send(NOTEMPTY);
     end PUT;
 end BOUNDED BUFFER;
```

Figure 2-2. An Example of a Bounded Buffer in Modula

This would increase the task population. The consequences of this could be to increase the task switching overhead in the system and to use up more memory for task representations.

For such tasks a notion of expedited dispatching is essential. As Hilfinger has shown [3], frequently such tasks can be eliminated by the implementation, or simplified considerably. For the success of Ada in these systems such strategies should be used where possible.

Similarly, for interrupt handler tasks the full generality of Ada tasking is rarely going to be used. The main concern here is to reduce the time between the occurrence of an interrupt and the time when the matching accept body gains control. It is most important that this time be much smaller than the time for a normal entry call.

2.3 OVERLAY MANAGEMENT

Does the implementation support the use of memory partitions, overlays, swapping, and program segmentation? If so what mechanisms are available for detecting thrashing problems? Is the use of virtual memory supported? Can both data and code be overlayed?

In many embedded systems, there usually are certain procedures which are executed very rarely, for example, maintenance and diagnostic procedures or certain man-machine interface procedures. It is prohibitive to keep the code for such procedures in main memory all the time. Since they are executed rarely, it is acceptable to have them stored on disk and have them overlaid when needed.

One of the convenient schemes for providing overlays in Ada would be to provide a pragma (say OVERLAY) applicable to packages. The meaning of this pragma would be to incorporate all the code and data associated with a package in an overlay which could be resident on disk. This will allow the users to organize rarely used software on disk and have it automatically loaded when called.

2.4 FAULT TOLERANCE

An important requirement in many military systems is that of survivability. This generally implies that a certain degree of fault- tolerance and fault detection mechanisms be present in the application system or the underlying system. It is not completely possible to meet this requirement in users' code without support from the underlying hardware or the operating system. For example, if the hardware has no mechanism for detecting power failures then the software cannot detect them.

2.4.1 Watchdog Timers

If the hardware has a watchdog timer then does the implementation offer any support for it?

In most applications a watchdog timer is used to monitor the sanity of the software. Before entering a section of a code, the software might set the watchdog timer to a value corresponding to an upper limit of the expected execution time; on exit the timer will be reset. If the timer is not reset within the specified interval, then a fatal error or fault is signaled, which may cause some recovery or clean up action to be performed. This mechanism is typically used to detect deadlock or infinite loops.

2.4.2 Audit Trails

In case of fatal faults, does the implementation log the state of the machine?

It is necessary in many systems to determine "what really happened before the crash?". Logging the state of the machine on a continuous basis, by the runtime system or by the application, will be detrimental to the runtime performance of the system. Support from the implementa- tion could offer certain features to ease this task. For instance, one could indicate to the system to invoke certain subprograms or tasks when interrupts associated with such crashes occur.

2.4.3 Fault Detection

What fault related interrupts are recognized? Are there additional implementation defined exceptions for fault-detection?

In general all fault related interrupts, if any, should be recognized. There could be two causes for these interrupts: either a fault which is external to the system as a whole, e.g., power failure etc., or some attempt to perform an illegal operation. The interrupts of the first category should and can be treated as regular interrupts. This handler could either be provided by the runtime system or the user. In either case, a reasonable recovery action could be performed.

If an interrupt occurs as a result of an illegal operation, then it can be handled separately either in the runtime system or the user's code, or the runtime system can turn it into an exception. The decision to treat illegal operations as normal interrupts or exception depends on the nature of the fault or the operation which caused the fault. For example, in many machines a divide by zero causes an interrupt; the runtime system must raise CONSTRAINT ERROR as a result of this interrupt.

At this point we do not know of any fault detection requirements of JAMPS which could not be treated in this manner. If the occurrence of an external fault causes an interrupt it should be treated as such.

2.4.4 Deadlock Detection

Does the implementation detect deadlocks? If so how?

The presence of deadlocks is an undesirable feature, to be interpreted as a 'bug' in the application program. The only reason, then, such a facility might be useful is to serve as a debugging tool rather than a strict requirement. One of the ways an implementation might signal the presence of a deadlock might be through the presence of an exception or a software interrupt. (e.g., an entry call into a special task)

2.5 SECURITY

2.5.1 Memory Protection

Does the implementation provide for memory protection of code and data?

In general, this issue is not really very relevant in many embedded systems. Assuming that the compiler is correct, then one can safely make the assumption that the code and data are protected from each other as required in Ada. However, with the option of inserting machine code, or interfacing to assembler code this assumption will not always be true. If the underlying hardware provides protected areas of memory, then an implementation could locate the code generated and the data areas in such designated protection domains. This provides further safety, even in the presence of low-level code.

NOTE: This form of checking is often required by NSA guidelines.

2.6 INPUT/OUTPUT

This section deals mostly with the input/output requirements of the JAMPS application. Since it was originally implemented on top of UNIX and the I/O facilities were adequate for the application, most of the concerns are operating system oriented concerns.

2.6.1 File System

1. What are the maximum and minimum record sizes for disk I/O?

2. What are the naming conventions for files?

3. How is the association of peripheral devices with files established? Can there be more than one disk unit etc.? What meaning is associated with the term 'EXTERNAL'?

4. Are library routines available for comparing one file with another, for copying a file from one device to another?

5. Must files be contiguous?

Some of the key functions of JAMPS involve message preparation as well as message management. This second category of functions involves keeping track of the history of message preparation over long periods of time, and storage of information on permanent or long-term storage media. Hence, the concern for a suitable file system. Below we describe in detail the concerns raised above.

1. The maximum and minimum record sizes which can be stored on disk will determine, to a large extent, how management information associated with messages (including message contents) is to be represented. This will determine the complexity of the intermediate layers used for storage and retrieval of various high-level message data representation in terms of disk records. At this point, however, there are no apparent requirements and it seems that a reasonable number would be sufficient. It is important to know this in advance for any detailed design of these subfunctions

(i.e. conversion between disk representation and in-core representation).

2. Similarly, it is necessary to know the naming conventions for files. This includes:

   a. Number of characters in a file name - if there are limits on lengths of file names, then it may imply an upper limit on the number of files supported by the system.

   b. Do file names form a flat name space or a hierarchical name space - This may determine if the name space is flexible and extensible.

   c. Is there a notion of file extensions?

   d. Is there a notion of version numbers ?

   e. Is there a default scheme for file name access (e.g., if no version number is specified is it the latest version or the oldest, etc.) ?

3. How are storage devices such as disks drives, tapes drives, etc. 'mounted' as file systems? Since the environment is going to be devoid of an operating system per se, and TEXT IO does not address this functionality, this issue is of particular importance for several reasons: It might be necessary to bring JAMPS disks from other installations for processing, or old backup disks might be required to be mounted for some message preparation work.

4. In order to make backups of JAMPS disks it is necessary to have 'file copy' functions which will permit this to be implemented.

5. Whether or not files must be contiguous is purely performance related, since this allows disk access algorithms such as the 'elevator algorithm' (i.e., those which minimize the disk head travel times) to be considered. This knowledge is not essential, but it can be useful in predicting the disk access times.

2.6.2 Asynchronous I/O

1. Must data be moved from I/O buffer areas prior to manipulation?

2. Can user request that the control be returned to the calling module following an I/O request? Can another task be requested to receive control following completion of an I/O request?

3. Can user cancel I/O requests? Are timeouts detected for I/O requests?

4. How are priorities associated with I/O requests?

The need for asynchronous I/O arises in all those systems, where a large amount of data is to be transferred between several processes. JAMPS is likely to be one such system, where possibly large amounts of data will be transferred between disks, memory, and special devices such as the line printer, communication ports, etc. The need arises from two factors: the amount of copying performed is too large under conventional schemes, and the amount of blocking time associated with such I/O or data transfer requests is large. Clearly, the built-in TEXT IO facilities of Ada do not address this need.

1. Many times when data is stored in intermediate buffers, it may be desirable to get a pointer to the data in the buffer or the queue. This may save a copy if the required data may not be needed.

2. When an I/O request (a read or a write) is made, the calling task would not like to wait. This helps increase the throughput of the system. The I/O request can be completed in parallel to other activities of the calling task, especially if the caller has sent a file to be printed.

3. Since JAMPS is required to support a line printer for the printing of various messages and information associated with them, it is necessary to provide cancellation procedures for deleting files from the I/O queues, or request a time limit for the completion of I/O requests.

4. In a multi-user or multiprocessing system, it may be necessary to order the I/O request queues to provide serial and urgent access to the various devices.

2.6.3 Device Support

1. What low level I/O drivers are supported? Is there support for writing new drivers?

2. What are the effects of plugging/unplugging of peripherals?

3. What types of support are available for creating non-standard (unique) device drivers? Must unique device drivers be written in Ada?

4. Is there an I/O driver in the target runtime environment for communication with the host computer which can be used for data collection in real-time?

5. Is there support for binary I/O? What restrictions are there on the types that can be used for instantiating I/O packages?

In order to make JAMPS device independent, the underlying system must allow individual devices to be replaced by different ones, possibly having different characteristics such as interrupt vector locations, protocols, etc. In view of this, it should be easy to introduce new devices and provide interfaces with them from software.

2.6.4 Terminal And Screen I/O

1. Is there support for generic terminal operations that are translated appropriately for specific terminals? Does it do windows?

2. Does console monitor provide capabilities for clearing of console screens? Is there a quick response command? Can cursor motion be controlled by character position, word position, line position, and page? Are there facilities for deleting characters, words, lines, pages, joining or splitting of lines, character overwrites and insertions, string location, text selection and transfers?

One of the main functionalities required by JAMPS is that of providing sophisticated word processing operations to assist in the preparation of various messages. Since most of the word processing is meant to be interactive, the high level representations of various messages need to be supported. In terms of the user, he should be able to view the message representation continuously on the terminal screen in order to have a meaningful and productive interaction with the system. In particular, the terminal should be able to support multiple windows, multiple buffers, multiple files, and an integrated set of operations for manipulating such representations.

2.7 PRAGMAS AND REPRESENTATION ISSUES

2.7.1 Pragmas

2.7.1.1 Inline. This pragma is most useful in cases of short subprograms ('one-liners') which, for modularity reasons, have to be implemented as subprograms. Support for this pragma will encourage programmers to respect the integrity of an interface without concern for efficiency. Furthermore, in the presence of an optimizer, the inline expansion of subprograms may open the path for certain optimizations which may not otherwise be detected.

2.7.1.2 Interface. This feature is most useful for incorporating code written in a foreign language. Since JAMPS is targeted for a stand alone system, the only programs which could be incorporated would be operating system independent routines from the previous versions of JAMPS (written in C), or certain machine dependent assembler routines (for efficiency reasons). In order to make a better analysis of this requirement, one would have to determine the extent of C programs which will be incorporated from the previous versions of JAMPS, and analyze the nature of the interfaces more carefully. However, the interface to the assembler routines should be most useful in tuning the performance of the final system as well as providing additional support for low-level programming (e.g. device drivers).

2.7.1.3 MEMORY SIZE. (see 2.1.3)

2.7.1.4 Optimize. Any form of optimization would be most useful for improving the runtime performance of the final system.

2.7.1.5 Pack. Since JAMPS supports a large number of specialized word-processing functions dealing with character strings. it would be highly desirable to provide a packed representation for such strings for maximum space utilization.

2.7.1.6 Priority. Since most of the I/O and message handling functions would be required to be programmed in Ada (requiring the use of tasks), there is a foreseeable need to control the relative urgency of these tasks. For this reason the use of pragma PRIORITY should be supported.

2.7.1.7 Suppress. As mentioned before, since JAMPS would be providing several string and array manipulation procedures and functions, bound checks will be generated for such manipulations. Therefore, it is wise to require the suppression of range checks.

2.7.1.8 SYSTEM NAME. SYSTEM NAME is a named constant defined in the package SYSTEM. Its value is an implementation defined enumeration literal. Since JAMPS is an application involving many computers sending specialized messages, it can be argued that there would be a need to name the various JAMPS workstations. With this view, it might be a good idea to be able to use SYSTEM NAME for determining the identity of various computers, as well as defining the valid names (i.e. the enumeration type NAME).

2.7.2 Representation Clauses

1. What type of control is afforded by 'length clauses'?

2. What allocation scheme is used for access types (e.g. space allocated on the stack, dynamic allocation, fixed allocation)?

3. Will the objects in a collection always have the same length if the designated type is 'unconstrained array' or an 'unconstrained record type with discriminants'?

4. What restrictions are placed on the use of alignment clauses?

A length clause is normally used to specify certain attribute values for certain types and task types. For simple and composite types only the SIZE attribute can appear in the length clause. This specification places an upper bound on the number of bits for all objects of that type. It does not, however, specify the exact size of such objects. Along with the other representation specifications, it can be used to specify an exact size (in bits) for the objects of such types. Since this is an extremely target specific concern, its portability is suspect. It should, therefore, not be used for bit twiddling and such applications. A proper use of length clauses would be to control the sizes of heaps for access objects. We will demonstrate a 'proper use' of this feature in conjunction with other length specifications to allocate an exact amount of storage later in this section.

For access types the only length specifications allowed are for the attribute STORAGE SIZE. This puts a lower bound on the number of allocatable objects for such collections. In many applications, the maximum number of allocations for certain objects is known well in advance. Therefore, it is desirable to initialize the heap to have that many objects. This is a highly desirable feature which must be supported by all implementations, for it allows for good management of available memory.

For task types, the length specifications apply to the STORAGE SIZE attribute. Such specifications define the number of storage units for an activation of a task object of the given type. This, essentially, puts an upper limit on the task stack. It is not clear whether any useful purpose is served by this, since the storage considered part of a task is implementation specific. In the absence of such information, the portability of this feature is highly suspect, and not really required for applications requiring high degree of portability.

EXAMPLE: A proper use of length clause for specifying the number of objects in a collection.

```
type OBJECT PTR is access OBJECT;
-- not more than 1000 allocations of
-- OBJECT PTR will ever be needed.
for OBJECT-PTR'STORAGE SIZE use
             1000*(OBJECT'SIZE/SYSTEM.STORAGE UNIT);
-- This 'fixes' the heap to contain 1000 'OBJECTS'
```

2.8 MISCELLANEOUS FEATURES

The following areas have been identified as potential features of an Ada implementation.

2.8.1 Special Optimizations

Special optimizations can decrease the overhead imposed by the runtime system.

Of particular importance is the overhead due to the runtime constraint checks introduced in the generated code. This is required by the language. However, a sophisticated optimizer could, at times, determine that a runtime check would never fail. In such cases the runtime check may be omitted. In the most general case a sophisticated theorem prover is necessary to detect such a condition. For example, consider the following code fragment:

```
-- linear search with sentinel.
if N = A'LAST then
 raise error flag;
end if;

A(N+1) := X;
I := 1;
while A(I) /= X loop
  I := I + 1;
end loop;

if I < = N then
  FOUND;
else
  NOT FOUND;
end if;
```

In the evaluation of the indexed component A(I) in the test of the while condition, a runtime check is not really necessary. In this case the pragma SUPPRESS can be used by the programmer to suppress the runtime check.

2.8.2 Predefined Packages

In addition to the standard reusable packages (e.g. CALENDAR), the implementation can supply packages whose bodies have been written in some non-Ada language (e.g. assembly) for efficiency reasons. Alternatively, a provision for interfacing with code written in assembly could be very useful for time critical applications.

2.8.3 Timing Services

Timing services may include maintaining the clock and providing accurate measures of time intervals between events.

SECTION 3

EFFICIENCY AND IMPLEMENTATION ISSUES

This section examines the implementation and efficiency issues concerned with the requirements discussed in the previous section. No particular distinction is made as to whether these requirements or features are implemented as part of the application or the underlying implementation's runtime system.

3.1 MEMORY MANAGEMENT

3.1.1 Selective Linking

Selective linking is very beneficial to most application systems. The benefits resulting from this feature could be applied both for trimming the runtime system as well as the application system.

There are two ways to achieve this goal in Ada. We will illustrate these two methods by examples. Consider a general purpose package STACK as defined below:

```
package STACK is

     type STACK TYPE is private;

     procedure PUSH
             (S : in out STACK TYPE;
              E : in  ELEMENT);

     procedure POP
             (S : in out STACK TYPE;
              E : out ELEMENT);
     private
          ...
    end STACK;
```

In the first approach the two procedures of the package STACK can be compiled separately as shown below. In this manner all packages which use this package can refer to the procedures as required. However, since all the procedures are compiled separately, the object code is in separate subunits, therefore, only those procedures that have been explicitly referenced would be linked.

```
     package body STACK is

          procedure PUSH is separate;
          procedure POP  is separate;

     end STACK;
```

The other method is to compile the two procedures together, and leave it to the implementation to selectively link and load the code for the appropriate procedures. However, it is most desirable to follow the first alternative as far as possible since the desired effect can be obtained in the application domain. The runtime support software as implemented by the compiler writers should follow this rule. This would allow minimal runtime code to be loaded into the target system.

Neither of these strategies involves any overhead at runtime, but they do involve significant overhead during the linking/loading phase. It requires the computation of following the procedural dependencies to determine which procedures need to be linked in or loaded. It seems that it is worth paying this price for obtaining smaller program size.

3.1.2 Free Storage Management

The simplest choice for a free storage management scheme is to support the allocator new as provided for by the language and implement the generic procedure UNCHECKED DEALLOCATION. This is perhaps the simplest and the safest mechanism, both from the point of view of the application programmer as well as the implementor. From the implementor's point of view, no fancy algorithms or data structures need be provided. From the application programmer's point of view, it leaves the option of implementing any sophisticated garbage collection schemes built around these two basic primitives. Furthermore, if the two given operations are sufficiently simple and fast, it can be used in predicting the cost performance of the application as a whole.

3.1.3 Free Storage Monitoring

In order to get a certain degree of storage monitoring several options may be employed. Perhaps the simplest option is to introduce an additional exception (say HIGH WATER) when the allocated storage goes beyond a certain percentage that can be indicated to the implementation via a pragma. This has a certain drawback that all allocators must be surrounded by exception handlers.

An alternative approach is to introduce an additional attribute for each access type T in the program whose value is the percentage storage left in the system. This has the appeal of leaving it to the programmer to check for high water conditions, but may require code to check the attribute in many parts of the program.

This approach does not require too much overhead, except updating of a location associated with the heap for a particular collection. However, particular caution is recommended in the use and implementation of this option when garbage collection is done on-the-fly. Since the high water information is shared between the allocator, the garbage collector, and the user of this information, inconsistencies could arise in the value of this attribute.

3.1.4 Free Storage Structuring

This feature has several distinct possible uses in a typical application. First, this feature's main use is for more effective memory utilization: for each collection declared in the program, if a maximum or minimum required size is known, then one could pre-allocate that size. This means that a certain amount of memory will always be available for use.

If there is a common heap reserved for all the collections in the system, then several allocation and deallocation requests on this heap must be serialized; since several tasks may be involved in these requests. Furthermore, memory management becomes more complicated, since various requests may be for different sized blocks of memory. If a separate heap is reserved per collection, then different requests for different collections need no mutual exclusion, and also for each collection all the requests may be of fixed size.

All this translates to better and faster memory allocation and deallocation. One approach is to have the implementor provide an option which will allow a separate heap for each collection. The other possibility is to explicitly maintain free lists for each collection in the users' code, thus assuring that the required memory management capabilities are available. From an efficiency viewpoint either option is acceptable, but from a portability viewpoint it is perhaps better to implement this capability in the application code itself.

As an example we give a specification of such a package

```
generic
    SIZE : positive;
    type MY_TYPE is < > ;
    type ACCESS_TYPE is access MY_TYPE;
package ALLOCATOR is
    function ALLOCATE return ACCESS_TYPE;
    procedure DEALLOCATE(X : in out ACCESS_TYPE);
end ALLOCATOR;
```

Specification of a generic allocator.

This generic package takes three parameters: the size of the collection, the type of the elements in the heap, and the access type name for pointers to the cells. The body of this generic package would contain a declaration of a heap of elements MY TYPE with the size SIZE, and all allocations and deallocations are guarded by a monitor task. In fact, if one decides to implement memory allocation locally as described above, a third function could be added which returns the fraction of memory left in the given heap.

```
generic
    SIZE : positive;
    type MY_TYPE is < >;
    type ACCESS_TYPE is access MY_TYPE;
package ALLOCATOR is
    function ALLOCATE return ACCESS_TYPE;
    procedure DEALLOCATE(X : in out ACCESS_TYPE);
    function STORAGE_LEFT return positive;
end ALLOCATOR;
```

Generic allocator with storage monitoring.

3.1.5 Target Memory Control

The recommended implementation of this requirement is to support pragma MEMORY_SIZE with the following interpretation. MEMORY_SIZE is treated as the total memory available for the code and data areas of the compiled program. In particular, with a fixed sized program (code and static data) this implies that the dynamic data areas available can be manipulated.

An implementation must treat the existence of this pragma as a parameter to the interface between the linker/loader and the storage initialization modules of the runtime system. A simple implementation may be to have the linker/loader prepare a table of contents for the runtime system, including the memory size of each component.

This does not involve any runtime overhead, except perhaps during compile time or during system activation time.

3.1.6 Memory Limitations

There are no apparent implementation or efficiency problems with this requirement.

3.2 PROCESSOR MANAGEMENT

3.2.1 Size Of Task Population

This is totally implementation dependent, but in general we do not see any problems in the implementation of this requirement, unless the target machine is memory limited, or has only limited processing power.

3.2.2 Task Priorities

There are no obvious implementation difficulties with this requirement.

3.2.3 Task Dispatching

There are several distinct cases where expedited dispatching is desirable so that certain monitor and interrupt handler tasks can be implemented efficiently. It is extremely hard to enumerate all the cases where tasking does not involve the full generality of Ada tasks, but we describe two commonly used idioms, monitors and interrupt handlers. One example of a use of monitor task was described in section 2.2.3. A straightforward translation of such a task into a program which does not use a task, but uses signals or semaphores, was also described. An implementation could provide two possible options, a semaphore package or a monitor package.

A semaphore package specification (in Ada) could be provided by the implementors of the runtime system. The body of this package could be a highly optimized version written in the underlying machine language. This would encourage the users to construct monitor-like packages without the overhead of tasking. Such a package specification might look like:

```
generic
package SEMAPHORE_PACKAGE is
    procedure P;
    procedure V;
end SEMAPHORE PACKAGE;
```

For portability reasons an Ada source level version of this package might also be provided.

Similarly, the implementors could possibly provide a monitor package whose specification is written in Ada, while the body is written in the machine language avoiding any tasking or task switching overheads.

An interrupt handler task could similarly be optimized if the corresponding interrupt entry is not called by any software task, or if the control does not 'leak' out of the body of the accept statement. This may mean that the queue management associated with entries could be eliminated and certain state saving operations in the handling of the entry call could be avoided. It general, it is difficult (sometimes impossible) for an implementation to verify these assumptions. However, an implementation could follow one of two strategies: provide a pragma which indicates that one or more of these assumptions would never be violated, or provide predefined packages which ensure that it is impossible to violate these assumptions.

For example, if an interrupt handler task and its declaration are hidden inside the body of a package, then it is impossible to call the entry from outside the package. It is quite easy to check for any calls to the entry from within the package itself or from its visible interfaces. Either strategy is acceptable.

3.3 OVERLAY MANAGEMENT

The issue of overlay management in Ada requires careful study and research. We feel that this issue must ultimately be addressed. However, we are not currently in a position to specify how such a requirement might be met, and what is the best way to achieve it. This needs to be studied more carefully to see if there are no semantic ramifications to adding this feature to the runtime system.

3.4 FAULT TOLERANCE

3.4.1 Watchdog Timer

A useful watchdog time, which is not very difficult to use, could be built by using the runtime facilities and having the following properties. When the watchdog timer is called with a certain value, the runtime system transmits the identity of the calling task to the timer. If the watch dog timer is not reset within the appropriate interval, an interrupt handler task or an ordinary task could be called with the faulty task's identity. In this manner some action could be taken against this task (e.g. abort).

3.4.2 Audit Trail

Logging the machine state is nearly impossible to do in user code. In most cases, a low priority task (possibly another processor) could provide prioritized asynchronous I/O to a storage device. In this manner, a certain amount of high-level information from user tasks could periodically be sent across to the log file. In the event of a fatal fault, the implementation could send high priority log information to the log file, and try to flush out as much as possible of the pending log from other tasks. For any kind of meaningful log to be gathered some of the system services needed might be: time stamps on each log, identity of the sender, etc. Any such scheme implemented on the same resources as the application software and with minimal support from the underlying system would have to be a compromise. Thus, to be most effective, the logging scheme would have to be highly implementation specific.

3.4.3 Fault Detection

Ideally, there is no problem at all for faults caused by hardware errors, or an interrupt by software errors. However, if this interrupt is to be turned into an exception, then some support from the runtime system is needed. We outline one scenario in which this possibility might be handled. Consider the case when an instruction (e.g. divide) could possibly raise an interrupt under certain circumstances (e.g. divide by zero). Suppose it raises an interrupt called FOO. If this interrupt is non-maskable and of a high priority, then the runtime system could provide a handler (written in assembly language) which would find out the task which was active prior to the occurrence of the interrupt, and call the exception delivery routines of the runtime system to raise an exception (say FOO BAR) in the correct task.

It should be clear from this example that this cannot be implemented without the knowledge of the task which was active prior to the interrupt occurrence. Therefore, a pragma (say EQUATE INTERRUPT TO EXCEPTION) of two arguments, the interrupt name and the exception name. An implementation could provide a list of interrupts which could possibly be equated to exceptions declared in users' code. For certain fault detection and handling this is the most natural approach; recovery should be performed in the task which caused the fault.

3.4.4 Deadlock Detection

The problem of deadlock detection is perhaps more difficult, and can be performed only in very restricted circumstances, some of them being disabling of all interrupts. Since JAMPS would have quite a few devices which interact with the CPU via interrupts, it would be very expensive to detect deadlocks. For most such purposes, to avoid or break deadlocks, the use of a watchdog timer is recommended.

3.5 INPUT/OUTPUT

Input/Output is one area, which is very application specific, and it is hard to come up with an I/O package which will be suitable for most applications or even the same application in its different stages of evolution. Thus, it seems that as JAMPS requirements are likely to change, even the initial I/O system would be not be adequate. Therefore, the best solution is to require only those features from the runtime system which would be adequate to implement most, if not all, application requirements outlined before.

3.5.1 File System

For most flexibility, it is desirable to have a simple but flexible file system, which does not make any assumptions about extensions, versions, etc. Since file extensions, versions, etc. are techniques used to encode specific information about files and the various tools which manipulate them, it would be best if the file system did not assume any such conventions. These could restrict the use of the file system unless designed particularly for JAMPS. Even in this case it would be inadequate if JAMPS requirements and conventions were to change. Also preferable would be a tree-structured hierarchical naming scheme for files. This would permit the designer to organize his files in a better manner, and also to extend the name space in a structured manner. This requires that library routines be available for the creation of directories.

Furthermore, facilities need to be provided for mounting and dismounting of entire file systems. In other words, there should be support for associating file systems and data storage devices.

The implementation should also provide a detailed explanation of the operations of closing and opening a file. The main use for these operations is to establish an association between external file objects and internal file objects. However, these operations are sufficiently ambiguous. If, for example, a file is opened twice, (in different tasks, or differently nested blocks), then the language does not mention if the state of the two file objects is required to be the same or different. In most systems open and close serve as a caching function; an open copies an external file to an internal buffer area, and subsequent operations on the file are performed on this area. Upon closing a file all the transactions performed on this file are committed, i.e. the file is copied back to the disk and the intermediate area of memory is destroyed. This is very important from an efficiency viewpoint.

The main arguments for requiring these features from the runtime systems are that, since Ada I/O already provides some file system-related functions (such as create, delete, open, close, etc.), it is perhaps easier to implement this additional support at this level. Another alternative is to implement the entire file system in application code. This would ensure that the file system is highly suitable, but this unnecessarily duplicates the I/O facilities of the language.

3.5.2 Asynchronous I/O

Our view is that asynchronous I/O should not be part of the runtime system at all. Such requirements are highly application specific, and such functionality can be addressed efficiently enough in the application written in Ada using the existing low-level facilities and the tasking features. Therefore, nothing is gained by depending on the runtime system for such needs. Furthermore, one gains portability by implementing such functionality directly.

We describe how to implement, asynchronously, transfer of large amounts of data from a producer process to a consumer process. A simple shared data structure, such as a bounded buffer, is inadequate for this purpose for two reasons:

1. The amount of copying necessary for transfer of data, from the producer to the buffer and from the buffer to the consumer, is too inefficient for large chunks of data.

2. The waits for access to the buffer are limited to the duration of the copying operation. This may cause the producer to drop the data if the data production rates become large.

A simple scheme to overcome these difficulties is outlined next. The producer stores the data in areas called buckets. Each bucket is maintained in a special pool. When data arrives, a bucket is allocated (from the fixed pool) and filled in with data, and then a pointer to the data (bucket) is put into a queue. On the other side, the consumer retrieves a bucket from the queue and empties out its contents onto the disk, and finally returns the empty bucket to the pool of buckets. In this scheme both of the above-mentioned difficulties are overcome; the access times to the queues are limited to the time for copying pointers to and from the queue, and the only copying operations are performed outside the queue. The overall architecture of the system looks like this:

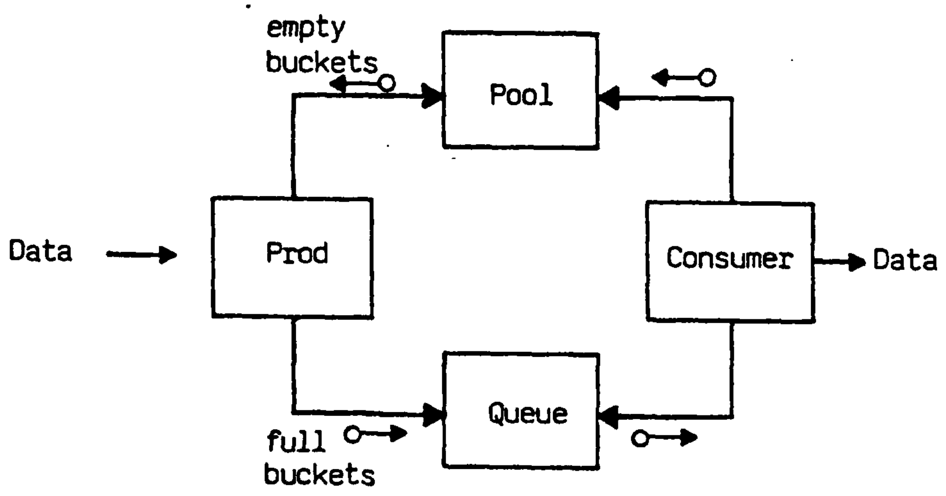


Figure 3-1. Data Flow Diagram of the System

With this scheme in mind, we get the following abstract structure of the producer and consumer processes.

```
Producer:
loop -- forever
-- 1. get an empty bucket from the bucket pool
-- 2. wait for data arrival
-- 3. fill bucket with the data
-- 4. put the bucket into the queue
end loop;

Consumer:
loop -- forever
-- 1. get a bucket from the queue
-- 2. empty the bucket onto the disk.
-- 3. put the empty bucket back into the pool
end loop;
```

Top Level Design of the Producer and Consumer Tasks

In this design, step 1 performed by Producer, and step 3 performed by Consumer need a mutual exclusion since they both access the common bucket pool. Similarly, step 4 of Producer and step 1 of Consumer need mutual exclusion since they both manipulate the common queue. However, the steps of filling and emptying the buckets can be performed independently, thus decoupling the two processes significantly.

One can derive a variety of asynchronous data transfer schemes from this simple model: by replacing the queue module by a module which implements a priority queue one can implement a line printer spooler. Similarly by tuning the bucket allocation algorithms in the bucket pool one can change the performance. Since the queue package can be implemented as a monitor task containing entries for appending and removing buckets to the actual queue, one can make timed entry calls and associate time outs with I/O requests.

3.5.3 Device Support

Most of the device support requirements can be supported via the combination of low-level features, representation specifications, tasking and unchecked programming. The nice feature of this approach, (to be illustrated below) is that it makes assembly level programming unnecessary. It also makes such programs more portable.

In this part we describe how new device drivers can be written using address clauses, representation specifications, unchecked type conversion, and tasks. Imagine a device which is attached to a keyboard, and upon the arrival of a character an interrupt is signaled to the CPU, and the character is placed at a location (a byte) in the memory. The following program describes the driver which returns characters from the keyboard.

```
package KEY_BOARD_IO is
   function GET_CHAR return CHARACTER;
end KEYBOARD_IO;

package body KEY_BOARD_IO is
    CHARACTER_QUEUE      : QUEUE_TASK;
    CHARACTER_LOCATION : CHARACTER;
    for CHARACTER_LOCATION use at ...;

    task KEY_BOARD HANDLER is
        entry KEY_BOARD_INTERRUPT;
        for KEY_BOARD_INTERRUPT use at KEY_BOARD_VECTOR;
    end KEYBOARD_HANDLER;

    task body KEY_BOARD_HANDLER is
    begin
        loop
            accept KEY BOARD INTERRUPT do
                -- put contents of CHARACTER_LOCATION into
                -- QUEUE_TASK
            end KEY_BOARD_INTERRUPT;
        end loop;
    end KEY_BOARD_HANDLER;

    function GET_CHAR return CHARACTER is
    begin
        -- return the first character from the QUEUE_TASK
    end GET_CHAR;
    end KEY_BOARD_IO;
```

3.5.4 Terminal and Screen I/O

Since most of these requirements are highly word-processing oriented and only one of the applications to be programmed in Ada, it is highly inappropriate that such requirements be supported by the language, or its runtime system, when there exists no such standard set of operations which would be useful in similar applications. The only rationale for requiring these facilities to be part of the runtime system is only that it might be more efficient, and convenient from the application programmer's viewpoint. However, since Ada does support low level facilities for programming hardware devices, and interfacing with assembly language routines, it can be implemented as efficiently in users' code. Furthermore, in the absence of strict requirements for such functions this task should be undertaken by the application designer who is more familiar with the overall application and can tailor the system accordingly.

SECTION 4

GUIDELINES FOR MINIMUM RUNTIME ENVIRONMENT FEATURES

In this section we review the features discussed above and propose some guidelines for the Ada implementors. For those features which, for practical reasons, are essential in application designers' design we propose to the implementors to provide such features as part of the implementation.

4.1 MEMORY MANAGEMENT

4.1.1 Selective Linking

1. Selective linking should be supported by the Ada implementor. For all compilation units compiled as subunits of a package specification or a package body, it should be possible to link in the directly or transitively referenced subunits only.

2. It is also recommended that the runtime system, if written in Ada, should be structured in terms of subunits as far as possible. As a measure of the support for this feature, the implementor should indicate the smallest and the largest amount of space that could possibly be occupied by the runtime system.

3. The implementor should also provide a reference guide describing the limitations of the selective linking mechanism, and explaining the most effective ways of using these mechanisms.

4.1.2 Free Storage Management

1. The language defined allocator must be supported by the implementation.

2. Similarly, the generic procedure UNCHECKED_DEALLOCATION must be provided.

3. Automatic garbage collection is not required, but if it is supported, then the pragma CONTROLLED must be provided as defined in LRM.

4.1.3 Free Storage Monitoring

1. Some free storage structuring capabilities would be preferred, though not strictly necessary. If this feature is supported, then one suitable mechanism is to provide an additional attribute which indicates the storage associated with a particular collection, or simply a function which returns the size of the entire heap if there is a single 'amorphous' heap.

4.1.4 Free Storage Structuring

1. A single heap is acceptable. If a separate heap is desired for one or more collections it can be easily implemented in the application code.

4.1.5 Target Memory Control

1. The implementor must provide for a mechanism for controlling the target memory size. The preferred mechanism is the pragma MEMORY_SIZE, however the documentation must describe this capability, and how it can be controlled.

4.1.6 Addressing Limitations

1. At this point we are not in a position to impose guidelines regarding the addressing limitations required for subprograms, packages, and tasks. The reason for this is that we do not yet have any definite 'feel' for the numbers which would be required by the final JAMPS application code.

4.2 PROCESSOR MANAGEMENT

4.2.1 Size of Task Population

1. The current design of JAMPS requires about 20 active processes. Since most of the common shared structures have to be represented as tasks it is recommended that an Ada implementation allow at least 60 active tasks.

4.2.2 Task Priorities

1. The implementation must support at least 10 levels of priorities, or alternatively allow the priority range to be redefined with at least 10 levels.

2. The scheduling rules for task of undefined priority should be clearly described.

4.2.3 Task Dispatching

1. It is recommended that the typical task switching times be comparable or better than the current task process switching times available to the UNIX* implementation.

2. Special optimizations for interrupt handling tasks should be provided. Especially useful would be the elimination of interrupt handling tasks.

3. Should provide a predefined library package which implements a semaphore or signaling scheme. The body of such a package should be written in assembly language, but an equivalent Ada source should also be provided for portability reasons.

4.3 OVERLAY MANAGEMENT

No special guidelines.

4.4 FAULT TOLERANCE

1. The implementation should recognize all fault related interrupts.

2. Some support for a watchdog timer should be provided.

3. There should be some mechanism for equating interrupts with exceptions.

*UNIX is a trademark of Bell Laboratories.

4.5 SECURITY

none

4.6 INPUT/OUTPUT

4.6.1 File System

1. Should support character string oriented file names, without any explicit notion of extensions or versions.
2. The length of file name should not be unreasonably small (e.g. 15 characters would be considered reasonable).
3. Should provide a tree-structured hierarchical naming structure for organizing files.
4. Should provide operations for creating and deleting nodes or directories in the tree-structures.
5. Should provide operations for searching and listing files in the nodes of the tree structured system.
6. Should provide operations for mounting and dismounting file systems (i.e. associating peripheral devices and files).

4.6.2 Asynchronous I/O

none

4.6.3 Device Support

The implementation should support all interrupts necessary to interface with the devices, and all representation and address clauses should be supported together with unchecked conversion for all types supported by the implementation.

4.6.4 Terminal and Screen I/O

Since terminals are treated as devices which can be interfaced in the regular manner no special support is necessary.

4.7 PRAGMAS AND REPRESENTATION ISSUES

4.7.1 Pragmas

1. The implementation must support the *pragma* INLINE.

2. *pragma* INTERFACE should be provided for the target computer's assembly language. In addition the documentation must indicate the parameter passing convention and the types to which parameter passing is limited.

3. *pragma* MEMORY_SIZE should be supported.

4. *pragma* OPTIMIZE should be provided.

5. *pragma* PACK should be supported.

6. *pragma* PRIORITY should be provided and the range of priorities must be at least 10.

7. *pragma* SUPPRESS should be supported.

8. *pragma* SYSTEM_NAME should be supported and the type declaration of SYSTEM.NAME should be modifiable.

4.7.2 Representation Clauses

1. An implementation must support address clauses for named objects (constants and variables), and entries. The implementation need not support address clauses for package objects, subprogram objects, and task objects.

2. Representation specifications for types must be supported. This includes: *pragma* PACK, length clauses, enumeration representation clauses, and record representation clauses.

SECTION 5

CONCLUSIONS

In this report we have presented some of the guidelines for runtime features necessary for the design of JAMPS. We have refrained from making any guidelines which suggest any minimum or maximum capacity requirements on Ada implementations. We do realize that without these guidelines it would be very difficult to select a suitable Ada runtime environment. However, without a detailed design for JAMPS, any figure or guideline we suggest could be meaningless. Thus a necessary step in the acquisition of an Ada runtime environment for JAMPS would be to arrive at a sufficiently detailed design. This Ada design, possibly expressed in an Ada PDL, could give a much more reasonable estimate for the required capacity of the runtime system.

A detailed design effort for JAMPS may also quite possibly expose some new requirements and a solution attempt may impose some further guidelines regarding certain features.

LIST OF REFERENCES

[1] Reference Manual for the Ada Programming Language, U.S. DoD 1982

[2] Grover, V., Rajeev, S., "Notes on the Ada Runtime Kit", SofTech Technical Report 9074-4, 1983.

[3] Hilfinger, P. N., "Implementation Strategies for Ada Tasking Idioms", Proceedings of the AdaTEC Conference on Ada, Arlington, VA, October 1982.

[4] Howe, R. G., "A Study of the Feasibility of Duplicating JAMPS Applications in the Ada Programming Language", MITRE Technical Report MTR 9167, MITRE Corp., Bedford, MA, January 1984. ESD-TR-84-160, ADA140884.

[5] Kamrad, J. Michael, "Runtime Organization for the Ada Language System Programs", Ada Letters, Vol. 3, No. 3, 1983.

[6] Lomuto, N., "Options in Ada Implementations", SofTech Technical Report 9074-2, 1983.

[7] Wirth, N., "Programming in Modula", Springer-Verlag, 1983.

DISTRIBUTION LIST

INTERNAL

D10

A. J. Tachmindji

D-36

J. B. Glore

D-45

G. A. Huff

D-46

S. M. Maciorowski

D-60

J. W. Shay
N. E. Bolen

D-63

G. Knapp

D-65

J. H. Galia
S. W. Tavan

D-66

R. L. Chagnon

D-67

C. J. Carter
D. P Crowsen
H. C. Floyd
E. J. Hammond
G. E. Hastings

D-67 (Continued)

R. G. Howe (10)
G. S. Maday
R. L. Micol
R. W. Miller
P. L. Mintz
C. D. Poindexter
S. M. Rauseo
D. A. Spaeth
E. J. Tefft

D-70

E. L. Lafferty
D. A. MacQueen
R. Sylvester

D-73

J. A. Clapp
S. J. Cohen
W. W. Farr
M. Gerhardt
E. C. Grund
R. L. Hamilton
M. Hazel
R. F. Hilliard
S. D. Litvintchouk
D. G. Miller
R. G. Munck
C. J. Righini
T. F. Saunders
K. A. Younger

D-75

R. T. Jordan
M. M. Zuk

DISTRIBUTION LIST (Continued)

INTERNAL (Continued)

D-77

J. M. Apicco
G. R. Lacroix
A. Sateriale

D-101

E. K. Kriegel
J. Riatta
L. C. Scannell
K. Zeh

PROJECT

Electronic Systems Division
Hanscom Air Force Base
Bedford, MA 01731

TCRB

B. J. Hopkins
Lt. J. Graves
Lt. M. K. Paniszczyn

ALSE

W. Letendre
Lt. A. Steadman
Lt. M. Ziemba

Langley AFB
Hampton, VA 23665

TAC/TAFIG

Lt. Col. E. Masek

EXTERNAL

Air Force Armament Laboratory
Eglin AFB, FL 32542

C. M. Anderson

Air Force Space Division
Directorate of Computer Resources
Box 92960 Worldway Postal Center
Los Angeles, CA 90009

Lt. Col. E. Koss, Director

Army Deputy Director
Ada/STARS Joint Program Office
3D-139 (400 AN) Pentagon
Washington, DC 20301

Lt. Col. R. Stanley, USA

Boston University
College of Engineering
110 Cummington Street
Boston, MA 02215

Dr. M. Ruane
Dr. R. Vidale

Language Control Facility
Wright Patterson AFB
Dayton, OH

G. Castor
P. Knoop

Naval Ocean Systems Center
Code 423
San Diego, CA 92152

H. Mumm

DISTRIBUTION LIST (Concluded)

EXTERNAL (Concluded)

WIS Program Office
The MITRE Corporation
D Building
Burlington Road
Bedford, MA 01730

Maj. R. Davis
Capt. P. Saunders

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02254

Ms. C. Braun (15)

Defense Technical Information Ctr
Cameron Station
Alexandria, VA 22314 (12)

AFGL/SULL
Research Library
Hanscom AFB, MA 01731